Skill-based framework description

Deliverable 3.1

Deliverable Title	D3.1 Skill-based framework description
Deliverable Lead:	TECNALIA
Related Work Package:	WP3: Rapid, intuitive task programming
Related Task(s):	T3.1: Skill-based framework
Author(s):	Hector Herrero (TECN) Iñigo Moreno (TECN) Anthony Remazeilles (TECN)
Dissemination Level:	Public
Due Submission Date:	28/02/2023
Actual Submission:	30/03/2023
Project Number	101017089
Instrument:	Research and innovation action
Start Date of Project:	01.01.2021
Duration:	51 months

Abstract	In TraceBot project TECNALIA proposed a common development framework, principally, for harmonizing developments and reducing integration efforts. The proposed Skill Framework has been shared with the partners and until now has been widely used by many partners.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089), 2021-2025

Versioning and Contribution History

Version	Date	Modified by	Modification reason
v.01	09.02.2023	Hector Herrero (TECN)	First version
v.02	10.02.2023	Anthony Remazeilles (TECN)	Document overview
v.03	13.02.2023	Hector Herrero, Anthony Remazeilles (TECN)	Ready for internal revision
V.04	15.02.2023	Julie Dumora (CEA)	First internal revision
V.05	22.02.2023	Hector Herrero, Anthony Remazeilles (TECN)	Adjustments based on revision
V.06	23.02.2023	Jean Baptiste Weibel (TUW)	2nd internal revision
V.07	24.02.2023	Hector Herrero, Anthony Remazeilles (TECN)	Revised version ready for submission
V.08	24.03.2023	Héctor Herrero (TECN)	Public version without confidential information

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

Table of Contents

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

1 Executive Summary

The programming of advanced and complex behaviours in robotics, like in other fields, is facilitated using frameworks of communication and abstraction. ROS is a significant step in that direction as it provides an off-the-shelf communication mean, as well as an encapsulation of operations mechanism through its concept of node (ROS processing unit). Nevertheless, the design of a robust, traceable and failure-resistant robotic application requires more layers on the top of the material provided by this framework.

We are proposing in TraceBot to use an implementation of a skill-framework to fill that gap. In such concept a robot is considered to be provided with a set of skills, as high-level capabilities or behaviours that can be combined to build applications. We rely on a previous implementation of Tecnalia of such concept, which provides (i) a model of skills, (ii) an execution manager that loads, orchestrates and monitors the good execution of the skills, and (iii) an application description format, that is used to define a process, as a succession of skills, which is loaded by the execution manager.

The advantage of this framework is that it is fully compatible with ROS and enables developers to implement skills using standard ROS communication interface (such as ROS actions and services). This is of major importance in collaborative projects with various partners, as it reduces the learning curve. Also, the hierarchical structure of the skills facilitates the encapsulation of functionalities and their reuse in multiple applications.

Several improvements have been provided to the framework with TraceBot, in particular to cope with the project requirements. The direct compatibility with ROS interface for skill implementation has been consolidated and is now the main implementation mean of skills in the project. The definition of an application, namely a process, has been completely upgraded to be based on the YAML format, a human readable format, which is also convenient to define sub-processes that can be reused into other applications. The execution manager has been updated to load the process as a state machine, which opens the door for implementing non purely sequential processes (where the next actions may depend on the outcome of the previous actions). Meta-information functionality has been integrated into the skill definition, enabling the association of contextual information or requirements with given skills, and opening the door to higher level reasoning. Also, the concept of skill has been upgraded to provide a systematic traceability of the operations conducted by the robot, through the publication of the actions configurations and outcome, which is then captured by the tracer to fill the audit trail.

The Skill framework is used to generate all implementations of the use case in TraceBot. Thanks to the compatibility with standard ROS actions and services partner can develop their modules (perception, verification, semantic reasoning, digital twin, arm and hand controller), and only have to provide entry points to their component, through ROS actions and services, to enable the integration of their component into the skill framework.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

2 Introduction

Developing robotic applications is hard, in particular when each new application may be different, use different hardware and be applied in different sectors and fields. These premises make it very difficult to robotize new applications that have so far been carried out manually, "forcing" companies to specialize in certain types of applications, hardware manufacturers and sectors.

A way to address this aspect consists in defining an abstraction mechanism, for decomposing a given application into a set of actions / operations. The usage of such framework presents many advantages. By introducing a concept of operation units, an action or a skill in our case, an application gets divided into a set of “high-level” operations, providing an overview of the different steps to achieve. This way it is easier to follow the activity planned, and to structure the visualization on the achievement progress. The encapsulation enables also to hide the complexity of the underlying operations. A simple motion to a given location may require complex environment perception and analysis, advanced path planning and cautious motion control to react to obstacles, but at the highest level, it can be defined as a simple “move to target”, relaxing at the highest level the need to understand all the underlying operations. Relying on such encapsulation mechanism also promotes the modularity and reuse of the implemented behaviours. With such frameworks, an interface contract is required to define how to structure a skill, how to parametrize it, how to get its outcome, etc… Therefore, a given behaviour can be seamlessly reused into another application, as long as the other application respects the generic interface contract and the specific requirement of the behaviour to be reused. Finally, another advantage of such abstraction scheme is that the change of a given behaviour or skill has limited impact on the overall application, as long as the interface contract is similar. It is therefore very beneficial for development stages as the overall application can be mounted and prepared even if all the behaviours are not yet totally ready. Going back to the “move to target”, a very preliminary version could be prepared, generating the motion assuming total liberty of motion, and used in early validation. Then, when the environment-aware planner and control system is ready, it can be seamlessly inserted in the process plan, replacing the previous basic version, without impacting the rest of the application if the interface is the same.

The usage of ROS in robotics is definitely a significant step into that direction, as this framework provides efficient means of encapsulating behaviours, into nodes, and transparent means of communication, through topic, services and actions. Nevertheless, it does not provide a systematic way to implement a complete application, as a succession of operations to be conducted, and it does not provide the manager that would be in charge to monitor the good execution of the process, and progressively triggers the execution of the next operation. This is why it is still relevant to build on the top of it a high-level framework to encapsulate behaviours, define a process of operations, and monitor its good execution.

Our proposition is responding to this need and relies on the *Skill-Based Programming* approach [Björkelund2011]. Robot skills are an analogy of human skills: a human can perform a known action in different situations without the need of re-learning how to do it. For example, for picking and placing an object the relevant information is the object to handle and the target to place, all the other needed information is inferred due to the prior learning and experience. The expected concept of skill for a robotic application can be presented as a software module that allows achieving a goal being flexible enough for adapting to various changes, such as, hardware devices, environment, locations, etc.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

Our implementation of a skill framework relies on three key components (see Figure 1):

- The core package is the Flexbotics Execution Manager. It is the engine responsible of loading a given process description (application defined through a set of skills), and orchestrating the progressive execution of the different skills of the plan, connecting the different input and output (parameters that each skill may require and results that may be generated, respectively), and configuring the skills as defined into the plan. The Execution Manager is agnostic to the purpose of the plan, and it can therefore be used for a very diverse range of robotic applications integrating, for example, advanced robotic manipulation tasks, computer vision operations, robot mobility and navigation, etc.
- The implementation of the so-called *skill,* which is an implementation of a capability, functionality. It can be related to perception processing, robotic motions, reasoning …. It must be designed in a standardized way so that it can be seamlessly triggered by the Execution Manager independently of its purpose and complexity. We took the decision to enable a direct compatibility with the standard ROS communication tools (in particular ROS services and actions), to reduce the knowledge required to provide skills, which is particularly relevant when working with external partners as it shortens the learning stage.
- The definition of an application description modality, describing the succession of operations or skills to be executed to produce the desired operation. We name it a *process*. The definition of the process is a combination of skills, where the connection of the skill input and output is defined, so that the Execution Manager knows at run-time which operation to trigger, with which parameters. The process definition provides a hierarchical scheme: a skill can be a basic unit (again, whatever its complexity), or a set of skills. This promotes again the reuse of developed skills, the abstraction of an operation complexity, and the decomposition of complex processes into a set of reusable operations. The process description is, thanks to TraceBot, now implemented through a YAML file, which has been selected for its readability and its ability to speed up process creation.

The rest of the document provides insight on the so-called Flexbotics skill framework (Section 3), and the improvements developed during the Tracebot project (Section 4).

Confidential information about the implementation has been removed from this document, but is available to the project consortium.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

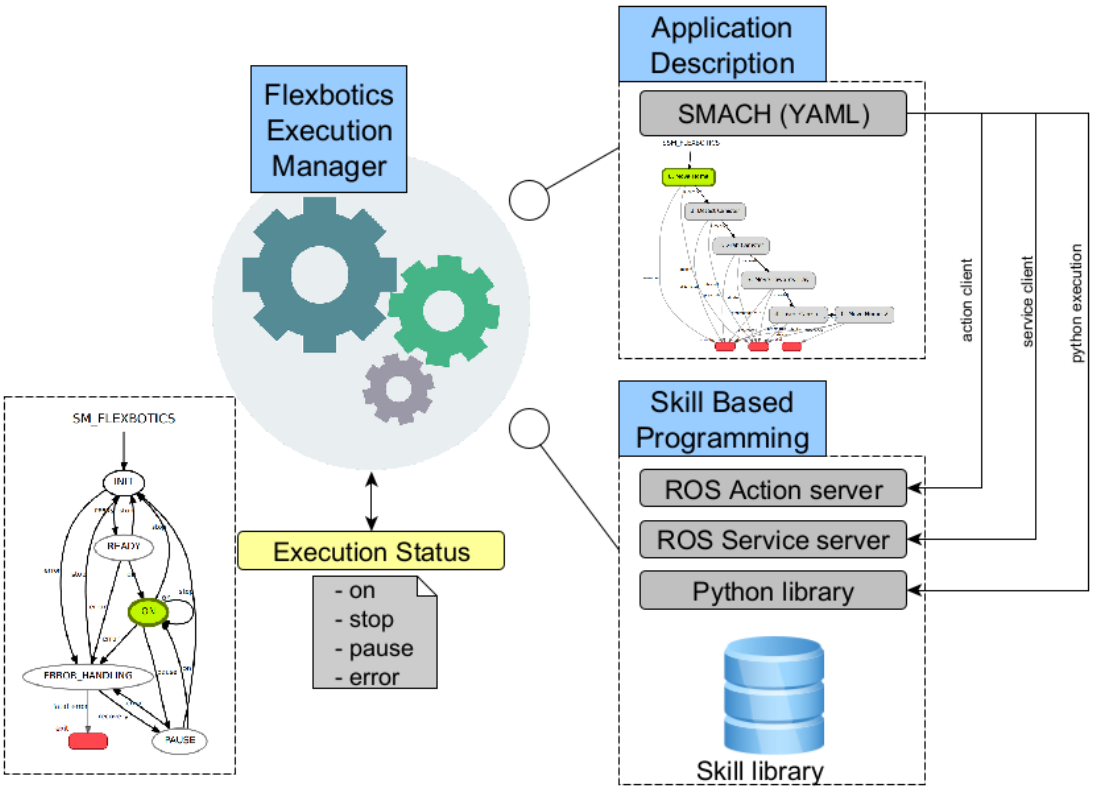

Figure 1 The Skill Framework is composed by three key components: Flexbotics Execution Manager, Skills and Application Description

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

3 Robot skills framework

For defining and encapsulating the required robotics capabilities, the *skill* concept is proposed. Having in mind the required efforts of integrating each module from different partners in different locations, the need to define a clear interface that must be fulfilled by any of the behaviours of the system is a crucial aspect.

In this section, the three key aspects mentioned above are described in detail: Section 3.1 introduces the concept of skills and their integration into Tracebot, Section 3.2 is focused on presenting how a robotic application can be modelled using YAML syntax, and Section 3.3 describes the details of the Flexbotics Execution Manager, the orchestrator that makes it work together. Figure 1 provides a general overview of the three main blocks of the Skill Framework.

3.1 Skill definition

The concept of *skills* refers to Skill Based Programming approach. Robot skills are an analogy of human skills: a human can perform a known action in different situations without the need to re-learn how to do it. For example, for picking and placing an object the relevant information is the object to handle and the target to place, all the other needed information is inferred from the prior learning and experience. The expected concept of skill for a robotic application can be presented as a software module that allows achieving a goal while being flexible enough to adapt to various changes, such as hardware devices, environment, locations, etc.

Despite the bibliography usually proposing a three-layer schema (primitives, skills, and tasks) [Pedersen2016], at TraceBot, the primitive concept (atomic or symbolic units of code) is not considered. The notion of atomic operation may be very different from one perspective to another. Also, it requires agreeing on programming practices at a very low level, which can be problematic for collaborative projects where partners may have quite different views on the matter. Therefore, TraceBot skills are considered as “black-boxes” that are responsible for achieving a goal, but this does not prevent from taking advantage of coupling skills to compose hierarchical skills. At the end, the skills can be sequenced forming tasks, which are mapped to the Sterility Testing use case[1].

To simplify the collaboration with partners, reducing the learning curve of the Flexbotics skill framework, we promoted the implementation of skills through standard ROS actions, a compatibility that was introduced in our skill framework.

3.1.1 TraceBot Skills

As mentioned above, a TraceBot skill can be wrapped into a ROS action server (see Figure 2). Thus, each contributor can develop the intended behaviour inside the action call-back, as usually done in the ROS environment. Then, for integrating the related operation as a skill in the Framework, only the following information is required and placed in the YAML process description:

[1] https://tracebot.gitlab.io/tracebot_showcase/

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

- Name. The desired name for the state which will wrap the action client.
- Action name: ROS path to the action.
- Action spec: ROS action goal message type
- Params: A set of pairs (key: value) with the required input parameters
- Result: Variable name where the result will be stored

In Figure 3 the YAML implementation of the action client can be seen. These blocks of code are interpreted by the Skill Framework for constructing the robot applications (more details at Section 3.2).

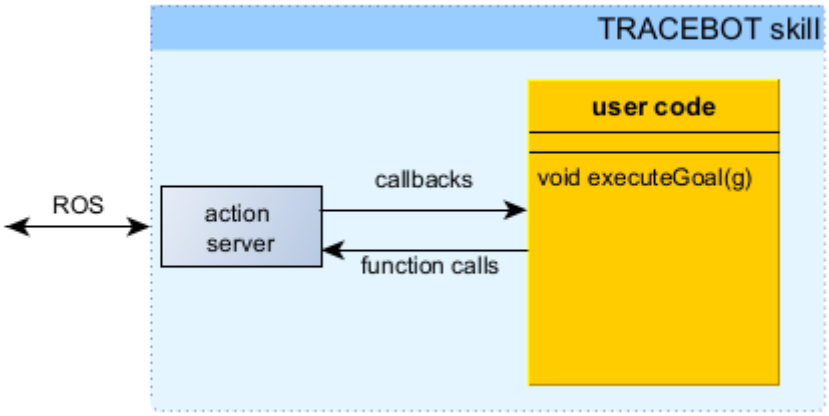

Figure 2: Schema of a TraceBot skill wrapped as a ROS action server

```
- !Action
  name: 1.1 Locate Canister
  action_name: /locate_object
  action_spec: tracebot_msgs.msg:LocateObjectAction
  params:
    object_to_locate: !Link object
  result: object_poses
```

Figure 3: Tracebot skill action client representation in YAML format

Additionally, the Skill Framework also allows integrating skills provided via ROS services, and Python classes. Services should be used for a remote procedure call that terminates quickly, since they cannot be pre-empted. Python skills are more intended for mathematical operations, data transformation, quick type management, etc.

3.1.2 Skill meta-information

The structure described in the previous section is sufficient to create a skill and embed it into the process. We nevertheless perceived the need to enhance the skill description with some contextual information that could be used to reason about the skills themselves. This is why we studied the option of incorporating some contextual or meta information associated to each skill.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

Meta-information is understood as a set of fields that complement the specification of a skill. Currently a meta-information file contains the following fields:

- Input Parameters
- Result
- Pre-conditions
- Hold-conditions
- Post-conditions

During the first period of TraceBot, it seemed interesting adding such meta-information directly in the YAML process files. Consequently, the Skill Framework was enhanced with the possibility of parsing YAML file with specific meta-information section for including the description of the skill (using appropriate languages such as PDDL [Fox2003]). This meta-information provides information regarding the required parameters of the skill, the preconditions to be fulfilled for being able to execute the skill and, finally, the effect in the environment of the executed skill.

Another approach that has been also implemented is the association of meta-information to each available skill within a separate file. This approach provides higher control and more re-usability, allowing to be managed by independent packages, and reducing the required occurrences of the meta-information if the skill is reused in other places. A ROS package manages the existing meta-information supercharged skills, providing auto-discovery features (capability to automatically identify the available skills at run time) and ROS services for accessing the meta-information. Thanks to *<export>* tag of the ROS packages the skill-metainformation can be exposed, and then detected by the skill meta-information manager. Through the mentioned services the available skills can be listed, and their fields can be retrieved from any node of the ROS application.

One of the important aspects to keep in mind when working with skills-based programming is the relationship of a skill with its predecessor and successor, i.e., its boundaries. It may not be enough providing its input parameters, since the skill could be responsible of executing complex behaviours and cause an effect in the environment; these conditions must be considered as well.

With the meta-information, a “contract” to be fulfilled can be defined: pre-conditions that must be satisfied to allow the execution of the skill, hold-conditions to be maintained during the skill execution, and post-conditions for setting an effect in the environment.

The candidate information for pre/hold/post conditions could be very extensive, thus as an initial approach the following fields has been designed:

- ROS communication requirements:
  - Required services
  - Required topics
- Resource status
  - Robot status
  - Gripper status

This information can be accessed via ROS services, and as a further work, could be managed by the *process checker* (see Section 4.2) in order to validate a process file before execution.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

The usage and configuration of a skill could be difficult for a non-expert. The parameter info can provide a guideline for an easier usage of the skills, allowing defining the type of the parameter (float, int, string, etc.), a default value, an informational field, and even a set of possible pre-defined values.

This information can be accessed via ROS services, being especially useful not only for the users, but also for developing more generic graphical user interfaces.

It is relevant to notice that such meta-information can be connected to the semantic information employed by the cognitive programming interface developed in task 3.6 to ease the programming of applications by the end-user, as well as to the ontology behind the KnowRob semantic framework developed in WP5. We are still analysing whether it is better to host the whole semantic information aside the skill description, or instead an identifier pointing to the related description in the complete ontology. In any case, the skill framework is able to cope with both implementation choice.

3.2 Process description through YAML syntax

A robotic application can be divided into steps, or more concretely, tasks. The philosophy behind the Skill-Based Programming consists on mapping the application steps with a hierarchy of skills that are able to accomplish tasks.

As previously mentioned, we implemented the functionality to define a process of skills through a YAML file, which required structure is now described.

3.2.1 Simplified Use Case process file

To illustrate the structure of such process file, we take as example the Simplified Use Case, selected for first stages of component integration at TraceBot, in particular during the first year of the project[2].

As illustrated by Figure 4, a succession of YAML tags are used. Each block of *!Include* tag refers to a set of skills included in another YAML file (increasing modularity and development reuse). Figure 5 presents the content of one of these included processes. The *!Python* tag indicates the usage of a *TraceableAction* (more details at Section 4.4). This refers to a ROS action that is executed in a remote action server, allowing the distribution of responsibilities and reducing the integration efforts.

Another important item is the communication between the skills. The input parameters and the generated results must be specified in the YAML process file. Each skill in the process file has a *params* field for defining the required parameters and, on the other hand, a *result* field for setting a variable name where the result will be stored.

At run-time, the Execution Manager is loading this process, and map it to a SMACH state machine[3]. To visualize generated behaviours, the smach_viewer[4] package can be used. It allows representing the

[2] https://www.youtube.com/watch?v=xAN3_wyX2xQ Illustration of the canister insertion process, as used for the first Milestone of the project.
[3] http://wiki.ros.org/smach
[4] http://wiki.ros.org/smach_viewer

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

process state machine generated by the Skill Framework, which creates different levels of hierarchy (for each *!Include* tag). In the Figure 6 three levels of hierarchy can be observed.

```
!Sequence
states:
  - !Include:tracebot_process:process/move_home.yml
    name: 0. Move Home

  - !Include:tracebot_process:process/detect_canister.yml
    name: 1. Detect Canister
    result: detect_canister_result

  - !Include:tracebot_process:process/grab_canister.yml
    name: 2. Grab Canister
    params:
      canister_pose: !Link detect_canister_result.canister_pose.object_poses[0]
      canister_pose_frame_id: !Link detect_canister_result.canister_pose.header.frame_id

  - !Include:tracebot_process:process/move_towards.yml
    name: 3. Move Towards Tray

  - !Include:tracebot_process:process/insert_canister.yml
    name: 4. Insert Canister

  - !Include:tracebot_process:process/move_home.yml
    name: 0. Move Home
```

Figure 4: Simplified Use Case YAML implementation of the process. The root definition is mainly referencing subprocesses defined in other YAML files

```
!Sequence
params:
  object: "Canister"
states:
  - !Python:tracebot_tracer:TraceableAction
    name: 1.1 Locate Canister
    action_name: /vision_locate_object
    action_spec: tracebot_msgs.msg:LocateObjectAction
    params:
      object_to_locate: !Link object
    result: canister_pose

  - !Python:tracebot_tracer:TraceableAction
    name: 1.2 Verify Canister Pose
    action_name: /dummy_server
    action_spec: tracebot_msgs.msg:DummyAction
    params:
      placeholder_goal: !Link object
    result: refined_canister_pose
```

Figure 5: Detail of the detect_canister.yaml process file, one of the blocks of the Simplified Use Case. Two ROS actions are progressively described. Both are defined as TraceAble actions, which is a wrapper defined in TraceBot to perform data traceability in a systematic way.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

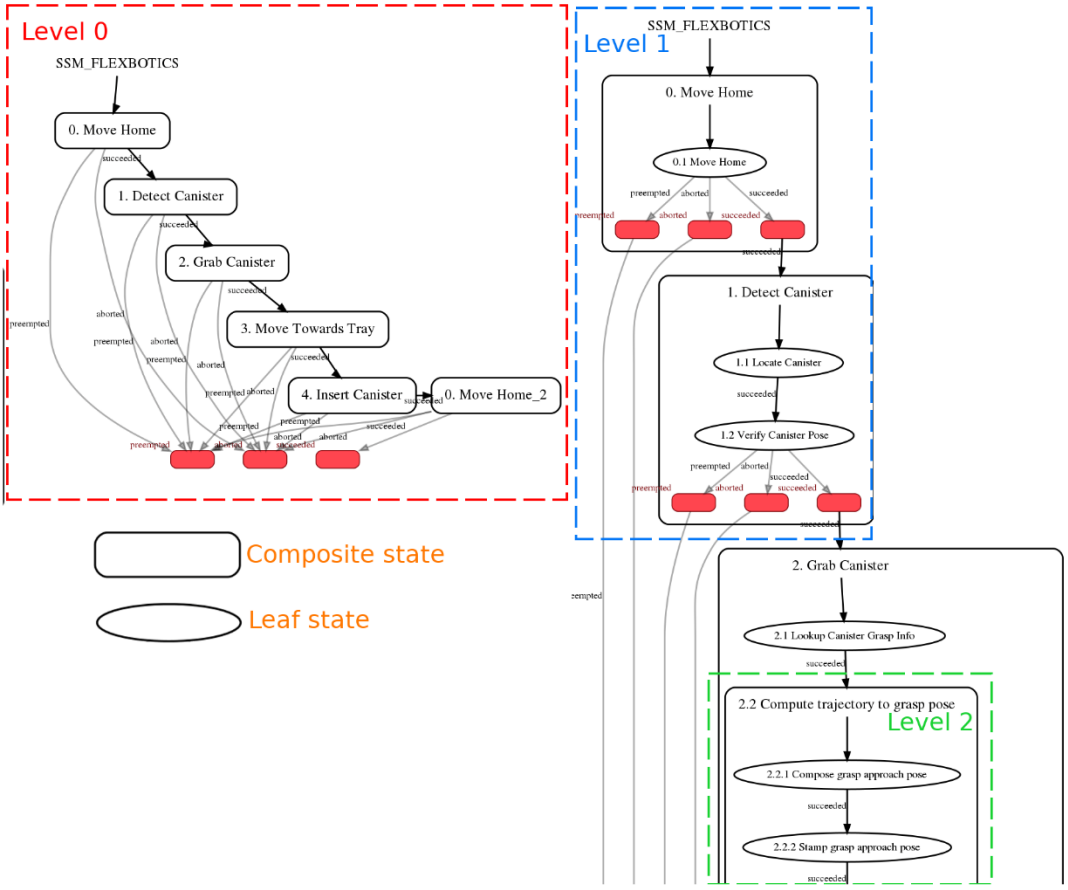

Figure 6 Hierarchical finite state machine generated by the Skill Execution Engine based on the sequences of behaviours described in the simplified use case process file "insert canister".

3.2.2 Available Flexbotics state classes

A complex application usually requires additional mechanisms beside a sequential execution. The Skill Framework supports conditionals and loops through decision states. Additionally, concurrence is also supported with a special state, allowing executing concurrently different operations.

In the current version the defined tags to construct the state machine and its states are: *!Link, !Sequence, !Concurrence, !Skill, !Service, !Action, !TopicCondition, !Decision, !SetVars, !Include, !Python*

- *!Sequence* and *!Concurrence* are state machines with a sequence of states. *!Sequence* executes each state one by one and *!Concurrence* executes them all at the same time. A state machine can also be included as a state to create nested state machines.
- *!Skill, !Service, !Action* are the main states of the state machine. *!Skill* calls a python function, *!Service* calls a ROS service and *!Action* calls a ROS action. They all store the results in their parent state machine. On the one hand, the actions should be used for everything that runs for a longer time than a couple of seconds, especially if robot or device movement is involved. Actions can be pre-empted and are designed for being able to provide feedback. On the other hand, services should be used for a remote procedure call that terminates quickly (e.g.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

perception routines, I/O operations, etc.), since they cannot be pre-empted. Additionally, for specific needs Python skills can be used. They are intended to use for mathematical operations, data transformation, quick type management, etc

- *!Link* tags are used to link input parameters with previously stored results. This enables using the result data from one state as an input of another state.
- *!SetVars* stores objects in the state machine directly from the YAML file. It can be used for index creation or initializations.
- *!TopicCondition* waits for a specific message to be published.
- *!Decision* are used to jump to another state if a condition is met. This allows to create non-linear state-machines. It can be used for conditionals and loops.
- *!Include* can be used to include another yaml file. Some of the fields of the included file can be overridden to allow reusability.
- *!Python* can be used to import arbitrary python objects. This tag also allows using SMACH states beyond the default ones.

Summarizing, an application is modelled in a file that we call process with a YAML syntax. This process file is a sequence of steps that we want to complete and each of the steps we call states having the analogy with the execution state machine that is generated from the YAML.

3.3 Finite State machine for execution control

The framework execution control is managed by the Flexbotics Execution Manager. It is built as a Finite State Machine itself. It manages the execution of the *process* files, which is loaded as a sub-state machine and represented as a succession of previously introduced *state* types (Section 3.2). The state machine allows simple and robust management of execution status such as resuming, stopping and error handling at every moment. The diagram presented in Figure 7 summarises the class structure of the Flexbotics Execution Manager.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

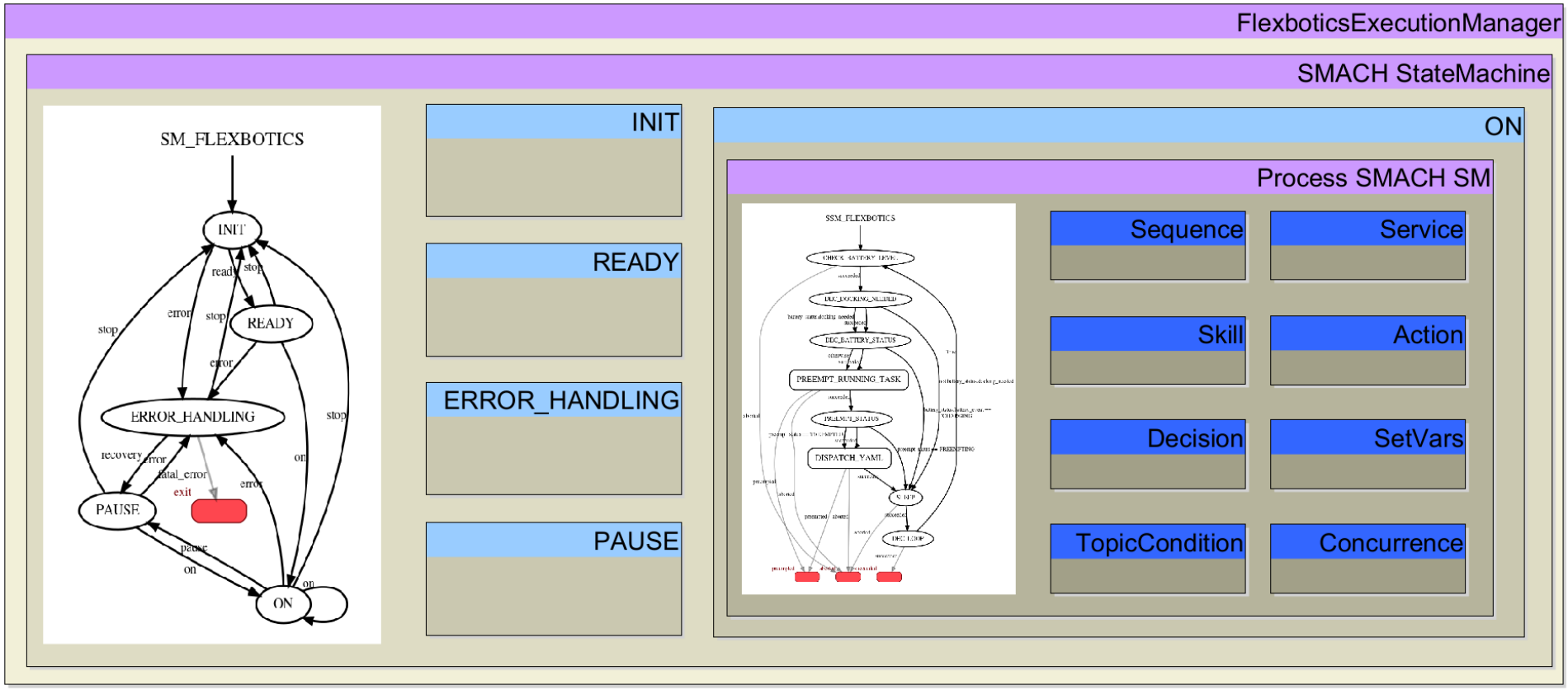

Figure 7: Flexbotics Execution Manager structure. Possible states of the Execution Manager are (light blue) INIT, READY, PAUSE, ERROR_HANDLING and ON. In ON state, the Flexbotics Execution Manager instantiates a sub-state machine from the process description using the indicated basic classes (in dark blue)

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089), 2021-2025

The Flexbotics Execution Manager can be in the following states (see left diagram of next figure):

- **INIT**: Responsible of initializing the process file to execute. When a process file is provided, it is parsed and checked to control its correctness: YAML process is well-formed, *!Link* tags correctly set, typos, etc.
- **READY**: Waits until `on` signal is received (usually when *play* button is pressed in the GUI).
- **ON**: *ON* state is the responsible of executing successively the provided process. The following bullets summarize the steps carried out by this *ON* state:

  1. Loads the previously validated YAML process description and creates a SMACH sub-state machine if it does not exist (such as the one presented on the right diagram of the next figure)
  2. Passes input parameters to the state
  3. Executes current state
  4. Stores state result values

- **PAUSE**: Waits until `on` signal is received without operating. If the `stop` signal is received, cancels ongoing execution and returns to INIT state
- **ERROR HANDLING:** Allows handling the errors that may occur during the execution. If an exception is raised the state-machine triggers to ERROR HANDLING state and there, depending of the application recovery strategies or controlled shutdown can be performed.

In Figure 8, the Flexbotics Execution Manager state machine and the application sub-state machine can be appreciated.

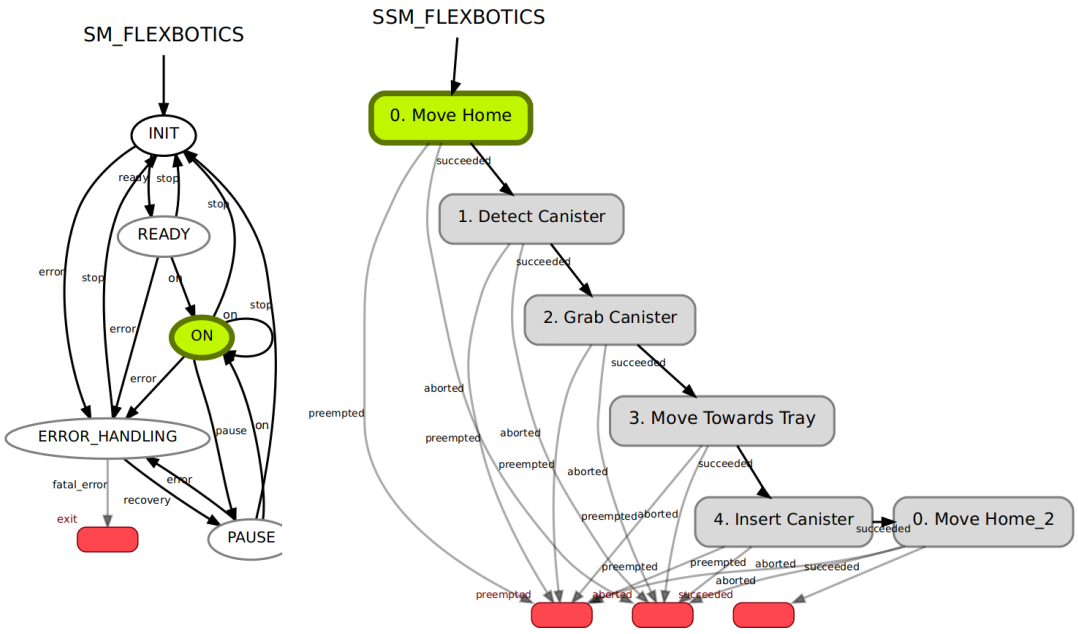

Figure 8 Flexbotics Execution Manager state machine (left) and the application sub-state machine (right). The right state machine is generic for any application, while the left state machine is specific to a process being executed.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089), 2021-2025

4 TraceBot improvements over initial Skill Framework

At the beginning of the Tracebot project TECNALIA proposed the Skill Framework as a common tool for robotic application development. Significant improvements were made in the early stages of the project and, throughout the project, new functionalities have been added according to the project needs. In addition, the partners have been supported to facilitate the use of the framework and to integrate their developments in an effective way.

4.1 State machine-based process execution

In the initial version of the Skill Framework, the process files were a sequential list of Python skills stored in a XML format. The Flexbotics Execution Manager executed them sequentially without any kind of possible logical conditions in the execution. Besides, there were no simple way for visualizing the current operation and the successive ones.

The development of a state machine-based process execution provides us with several advantages:

- More complex applications can be modelled (i. e. not necessarily sequential), thanks to the conditionals and loops logical operators that can be implemented. With the previous approach each skill must handle the error handling behaviours, thanks to the new implementation the skills can have smaller granularity and delegates the logic to the YAML process.
- Improved introspection features
- More traceability and robustness

4.2 Process checker

With the previous version of the Skill Framework, it was common spending significant time fixing bugs in the XML process files. There was no mechanism for checking the XML schema and syntax. In addition, since XML files were usually created by hand, introducing errors, typos and copy & pasted blocks of code that cause runtime errors was incredibly common.

The process checker is a utility package that allows the validation and standalone execution of YAML state machines. With the process checker implemented in TraceBot, the YAML process files can be checked at any time. Besides, the process checking has been incorporated to the process loading stage in order to avoid the majority of the runtime errors caused by typos or not consistent implementation of the process tasks.

It is available as a command line tool that can be called with `flexprocess`, similar to the ones provided by ROS to check messages and services. It has the following options:

- *flexprocess show*: Display the contents of the process file
- *flexprocess info*: Display information about the state machine, such as inputs parameters and results
- *flexprocess run*: Run the process file and show the outputs

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

- *flexprocess mock*: Run a mock version of the process file, which instead of running each state, just mocks the output values. This is useful to check the correct linking of the variables without having to run the whole process.

4.3 YAML syntax for process files

As mentioned above the previous implementation of the Skill Framework was based on XML process files for describing a robotic application. Both XML and YAML file can be easily parsed, but YAML has simpler syntax and enables a simple usage of the *tags*[5], a strong argument for changing the format. Through YAML tags application-specific classes can be referred directly from the YAML file. The tag mechanism provides a simple way for constructing SMACH state machines, with the required diversity of states, when the process files are loaded.

4.4 TraceableAction

The Tracer component developed as part of the TraceBot project is responsible for creating the final audit trail based on the process execution. The tracer component adds traceable information provided by the *!Action, !Service, !Skill*, and *!Sequence* tags. These are integrated into the skill engine using *!Python* tags, i.e. *!Python:tracebot_tracer:TraceableAction* instead of *!Action*. Using this tag, a inherited class is used instead of the standard skill class. This inherited class handles the publication of the input, output and possible feedback of the operations, information which is captured by the Tracer module to enrich the audit trail. This can be done automatically since all ROS data format are serializable, Thanks to this inheritance, the publishing process is made automatic so that a creator of a new action does not have to take care of this transmission, which enforces the mechanism usage at minimum cost.

More details on the Tracer component can be found in D4.1.

4.5 Meta-information

The initial implementation of the Skill Framework suffered of lack of cohesion between the skills. The skills could not be considered as independent and self-sufficient entities and, consequently, often additional pieces of code and checks had to be introduced inside the skill implementation.

The skill meta-information proposed in Tracebot intends to provide a common structure for defining the requirements and the effects of the skills. In that way, the implementation of the checks for required pre-conditions can be centralized in another entity outside the skill implementation. Similarly, the effects in the environment can be registered without having to adapt the implementation of the skill.

[5] https://yaml.org/spec/1.2.2/#24-tags

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

4.6 Skill framework usage in TraceBot

The skill framework is extensively used in the project, for the functionalities and the ease of integration it provides. All the robot applications are implemented using the framework. Each use case executed on the robotic system is currently described through YAML process files as presented in this document. Following the hierarchical description of the complete use case, accessible at the TraceBot showcase website[6], the processes are divided into subprocesses (i. .e. YAML files) that can be reused for other applications requiring similar behaviours.

The developers focus on the development of their module, from the perception and verification aspect (in WP3 and WP4), the hand controller (WP2) and arm controller (WP3), to the system reasoning with the KnowRob platform (WP5) and the Digital Twin representation (WP5). The appropriate entry points are jointly defined, and made available through standard ROS actions and services interfaces. These entry points are then introduced in the process files as skills using the tool described in this document. The advantage of this strategy is that the modularity capability, key characteristic of the ROS framework, is also maintained using the skill framework. This way we maintain all associated advantages, such as the ease of code sharing, deployment and upgrade.

The Skill Framework acts as an orchestrator, dispatching each of the skills in the process file to the responsible module. It also takes care of relating the skills to each other, ensuring the data flow from one skill to the other.

With such framework, the connection with the traceability and verification layer developed in WP4 is also facilitated. Verification components can be incorporated in the process description, using the same interfacing mechanism. The incorporation of the traceability concept at the level of the Skill framework, as described in this document presents the advantage of ensuring a systematic management of such matter in the complete flow. The implementation of the traceability concept is centralized in a well-localized piece of code (the TraceableAction class), so that any variation or adjustment of the implementation or concept can be done without having to refactor all the skills already implemented, which corresponds to a significant gain in time.

[6] https://tracebot.gitlab.io/tracebot_showcase/

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

5 Conclusion

In TraceBot, the Skill Framework has become the de facto integration and execution environment. This framework provides a standardised way of constructing and integrating each organization's contributions, thanks to a set of guidelines and best practices for developing skills. The development of different functionalities is approached using the skill concept, which comes from the robot Skill Based Programming philosophy. Each partner in the project has implemented its set of skills, wrapped as action servers within the ROS ecosystem. This distributed approach enables different partners to work independently but, thanks to the Skill Framework, the integration of the development is significantly simplified.

A YAML syntax is used to create a robot program using the proposed framework. These files, called processes, are, in short, a sequence of blocks with a specific label for each one of them. This document presents the tags available and their purpose. Furthermore, it also describes how SMACH state machines are now generated from these process files to execute the described behaviours. The Execution Manager is in charge of loading these process descriptions, and then taking care of the execution of the processes by controlling the configuration and execution of the described skills.

One of the main advantages of the framework is that it is generic enough to be applied in a variety of scenarios. Consequently, throughout the TraceBot project, several functionalities have been added according to known and emerging needs. One of the most relevant and widely used functionalities in the project are the TraceableActions, that are an extension of the original skill model to include some traceability functionality for automatically publishing the input, output and possible feedbacks of the operations, information that is captured by the Tracer module to enrich the audit trail.

The main functionalities of the skill framework are now implemented. Even though the task T3.1 in which that work has been conducted is getting finished, the framework could still be improved based on the additional needs that may be identified during the further experimentations.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)

6 References

[Björkelund2011] Björkelund, A., Edström, L., Haage, M., Malec, J., Nilsson, K., Nugues, P., & Bruyninckx, H. (2011). On the integration of skilled robot motions for productivity in manufacturing. In 2011 *IEEE International Symposium on Assembly and Manufacturing (ISAM)* (pp. 1-9).

[Fox2003] Fox, M., & Long, D. (2003). PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20, 61-124.

[Pedersen2016] Pedersen, M. R., Nalpantidis, L., Andersen, R. S., Schou, C., Bøgh, S., Krüger, V., & Madsen, O. (2016). Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*, 37, 282-291.

Horizon 2020

TraceBot receives funding from the European Union's H2020-EU.2.1.1. INDUSTRIAL LEADERSHIP programme (grant agreement No 101017089)